NETAVIS
Observer 4.0
Server Customizing

1. NETAVIS Observer 4.0 Server Customizing

Document version V1
Published in April 2011

2. Copyright

NETAVIS Software GmbH
Blindengasse 3
A-1080 Vienna
Austria

Tel. +43 1 503 1722
Fax. +43 1 503 1722 360
info@netavis.net
www.netavis.net

Contents

1. Introduction to customizing NETAVIS Observer

This document describes how you can customize various aspects of NETAVIS Observer. Depending on the part you want to customize, you will need different levels of knowledge and skills to securely make the desired changes. Some parts need UNIX knowledge, some XML or JAVA. In order to program or modify event handlers, you need to have basic programming skills, too.

1.1. The NETAVIS Observer documentation set

These documents are available:

- *NETAVIS Observer User Manual*
- *NETAVIS Observer Supported Video Sources*
- *NETAVIS Observer Server Installation and Administration*
- *NETAVIS Observer Server Distributed AS Administration*
- *NETAVIS Observer Server and Client Compatibility*
- *NETAVIS Observer Server Customizing*
- *NETAVIS Observer SNAP XML Interface*

All these documents are available online as PDFs directly on each NETAVIS Observer server via the standard web interface and also from each client via the **Info** menu at the lower right corner.

1.2. What can be customized

Customization in NETAVIS Observer is a dynamically expanding feature. Our goal is to give our customers all the power of NETAVIS Observer with the ease of flexibility. Therefore we continuously extend the palette of customizable components of NETAVIS Observer.

Currently Observer allows you to customize these elements:

- EMS event handlers: modifications
- External I/O plugins: enabling/disabling and definition of parameters

2. Observer Event Management System (EMS)

2.1. Introduction to EMS

Events are a central concept in Observer for documenting and distributing state changes in a running Observer service or server. Generally speaking, events are of various types, like motion detection events, camera failures, but also system events like user logon and logoff. Observer has a predefined set of standard events (see screenshot below). However, Observer also allows custom events for special applications.

Events are handled and managed by the **Event Management System (EMS)**, a powerful subsystem of Observer that handles events and triggers appropriate **actions**. For example, an action can be start

archiving, send an SMS or start a picture stream to a remote host. What actions are triggered and how is defined by so called **event handlers**. To enable system integrators to fine tune this wiring between events and actions, EMS offers to edit/modify these event handlers.

So called external I/O plugins serve as interface elements between the outside world (external devices, interfaces and external software) and the Observer EMS. These plugins handle the device specific communication and generate custom events. These custom events are then handled by so called custom events handlers. What parameters these custom events carry is up to the device specific code. Currently such plugins are created by NETAVIS Software but in one of the following releases we will open up all the interfaces (also the plugins) for external customization.

The **EMS is implemented in the JAVA programming language**, therefore all events, actions and handlers are encapsulated in **JAVA classes**. Users wanting to customize the EMS system have to be able to understand the JAVA language and the class inheritance principles what we will describe later in this document.

Handling of an event in the EMS is done using class inheritance. Each predefined event in Observer has an event class and a so-called handler class. Each handler class has the same name as the event class itself extended with the name "**Handler"**. Events are ordered into categories and sub-categories. They inherit from each other. This hierarchy is shown on picture 1. For example the event **MotionDetection** is a child of **InPictureEvent**, which in turn is a child of **CameraEvent**, and so on. To understand the concept of EMS, let's look at the flow of control.

When an event is received EMS first locates the most specialized handler class for the event. This is the class with the same name as the event (e.g. MotionDetectionHandler when the incoming event was a MotionDetection event).

The **handleEvent()** method of this handler class is then called with the event. When control comes back to the EMS it looks in the class inheritance tree for the parent of the original event (staying with our example it will be the InPictureEvent class) and passes the event to the corresponding handler class (InPictureEventHandler). This chain continues until EMS reaches the top of the event tree (the class Event). As you can see, the event travels through all handlers in direction from the most specific to the least specific. Each handler in turn can decide whether it should act (e.g. generate actions) to the given event or not. At the top of the tree (Event) a generic handler exists which is responsible for user notification, email, SMS sending, etc.

When the EMS has fed the event through all handlers, it looks into the contents of the event and checks whether it should be saved in the database, forwarded to other servers, etc. This is the last point in the handling chain of the event.

Now let's have a look at the method which is the heart of event handling. This is the above mentioned handleEvent method with the following signature:

```
public void handleEvent(Event event) throws Exception
```

This is the point where handler writers can place their piece of Java code to generate action to tailor system behaviour upon events.

Please note: Throughout this chapter we are talking intensively about event classes, handler methods, actions, etc. In the distribution you can find the full documentation of all the classes mentioned here. The file is called **EMSAPIdoc.zip** and can be downloaded from our web pages (Start **Customizer -> Download configuration files**). The content is a standard javadoc HTML documentation. After unpacking the zip file you can look at the contents using a web browser. Please consult it for the signature and usage of each class or method referred herein.

The main helper class (what does it mean?) is the static class **ActionPropagator**, which defines methods for creating and sending of the predefined actions. Each method creates an action, fills it out with the user supplied parameters and sends it to EMS for forwarding to listeners.

The documentation covers the **Declaration** class as well, which is the parent of all event classes. Using these methods you can query values stored in events and decide what is to be done.

2.1.1 Custom events

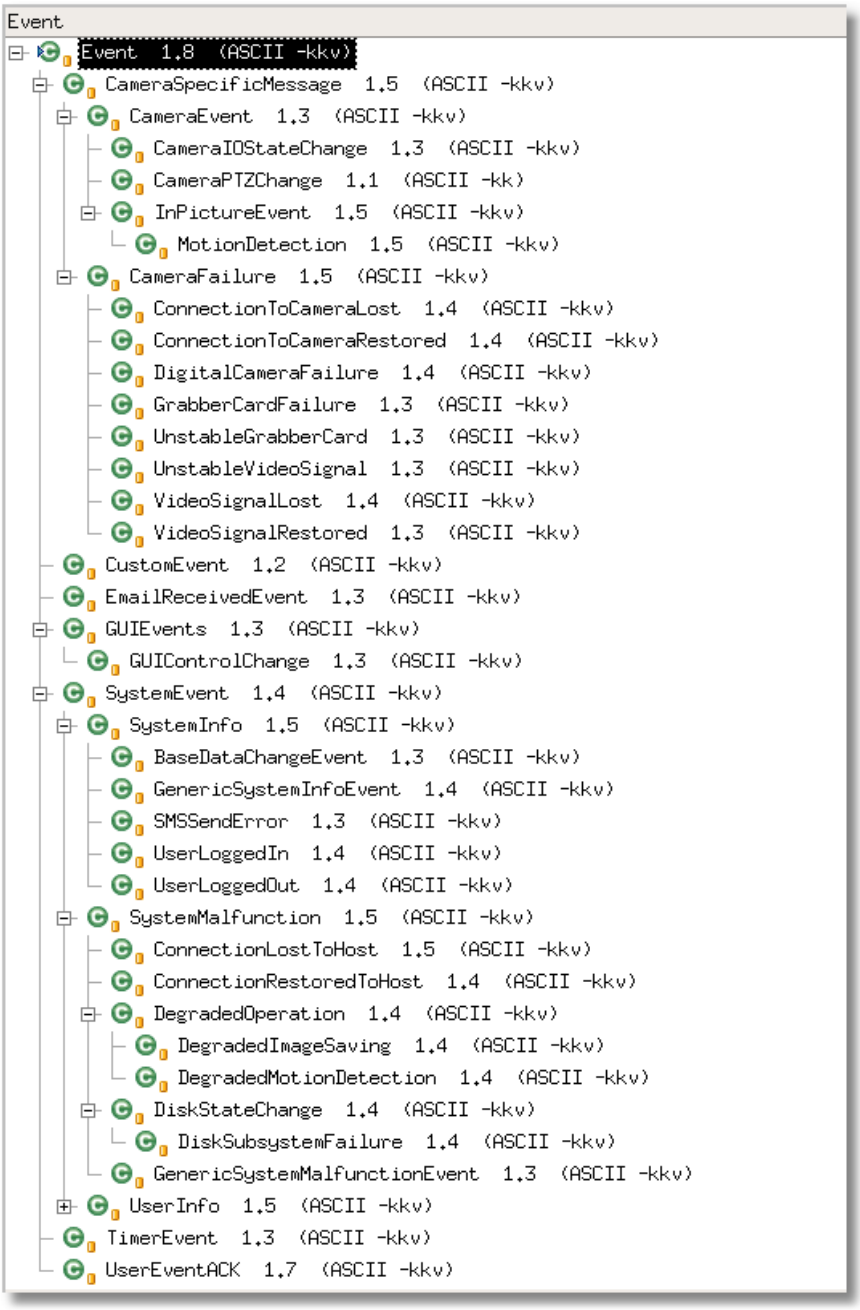
Apart from standard events, Observer provides a special event for third party event writers (e.g. working via the SNAP interface). Such a special event is called custom event (represented by class **CustomEvent**) which is similar to other events but its name and embedded instance variables are user definable. Such an event can have five String parameters. What is exactly stored in these parameters is up to the implementer. Of course, when numbers are stored as strings, the user has to convert them back to numbers before use.

Retrieving parameters from custom events differ from standard events. Five parameters are stored in the event all together (as Strings) and can be retrieved by calling the getParam1()..getParam5() methods respectively. The “name” of the event is stored in the CustomType field which is returned by the getCustomType() method (see Documentation of the class declaration in the EMS API documentation).

All custom event specific handlings should be put into the class **CustomEventHandler**. Because for all custom events there is only one handler the handler writer should split up code between different events by checking the type of the incoming event:

```
public void handleEvent(Event event) throws Exception {
    log(“CustomEvent event handler starts”);
    CustomEvent origEvent = (CustomEvent)event;
    if (origEvent.getCustomType().equals(“myEvent”)) {
        if (event.isEventLocal())
            handleMyEvent(origEvent);
        else
            log(“Event is remote – skip handler body”);
    }
    else if (…)
    …
    log(“CustomEvent event handler ends”);
}

private void handleMyEvent(CustomEvent event) {
    …
}
```



Picture 1. Hierarchy of event classes

2.2. Editing event handlers

Handlers can be edited in two ways:

Editing the files via the command line interface directly on the server

2.2.1 Editing the files directly on the server

We recommend that you edit and build your handler classes directly from the Linux system. Of course the web interface is also at your disposal.

If you decided to work directly from the operating system, please log in as **root**. When you are in the system you can find the full source code if the handler customization environment in the **/netavis/Distribution/custom** directory. The directory contains a makefile and a directory with the sources. For editing a handler you have to change to the proper directory.

```
cd /netavis/Distribution/custom/server/event_manager/handlers
```

For editing you can use **vi** (use it if you know the vi commands) or also **mc** (midnight commander, which has an interactive fullscreen editor) which is installed on each server machine. After finishing the changes, you have to compile the java files executing a **make**.

```
cd ../../..
make
```

All errors during compilation will be shown on the screen. Repeat editing and making until you are ready. During the make your classes are compiled, stored in **jar** (Java archives) files and copied into the **libs** directory of Observer. To test your handler you have to restart Observer:

```
cd /netavis/Distribution
./netavis.sh stop

… wait until the NETAVIS stopped …

./netavis.sh start
```

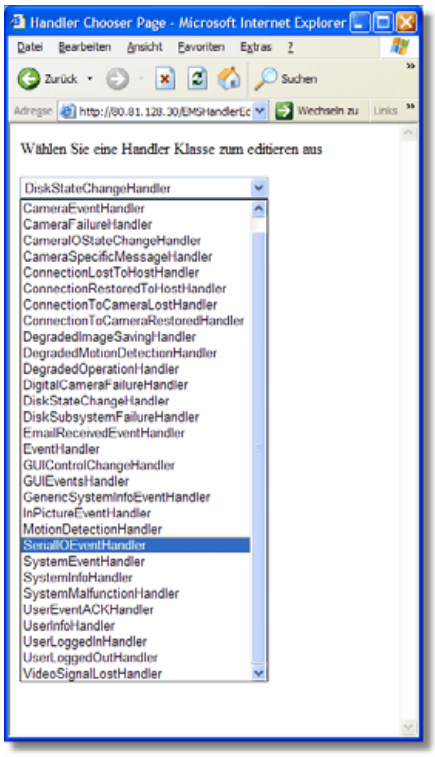
In your handler code you can use the documented **log()** method to emit debugging information. Each call to this method will write the message into the following file:

/netavis/NetAViS_out/logs/EMS_UserLevel_machine-name.log

Where machine-name is the name of your server machine. Inspecting this file after or during the run of your handler can help you in debugging your piece of code.

2.2.2 Editing handlers via the web interface

For editing the handlers you can also use the web interface. It the **Customizing plugins** chapter we showed you how you can find the customization web pages of Observer. In this page you will find the link to **Start handler editor**. To choose which handler you want to modify use the following drop down menu.



After clicking the **Edit Handler** button a new page is opened with the current content of the class. When you are ready with the modifications you have compile and active your code.

When you are ready with your modification click on the **Save** button. This will save the current contents back onto the server.

Abb. 2-1: Observer Handler Editor – Save the changes

When saving was successfully done the “save successful“ message will appear. (see ???REF).

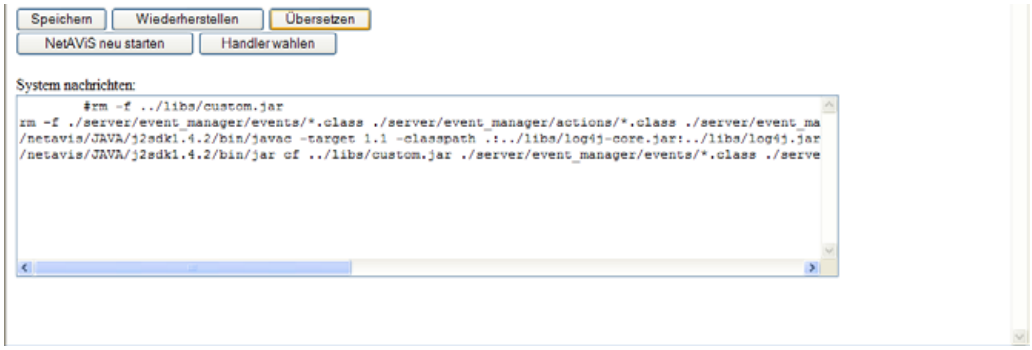


Abb. 2-2: Observer Handler Editor – Compiling the handler

After saving the file you have to compile it. Click on the **Compile** button to start the Java compiler. All messages from the compiler will be shown in the **System messages** text box. When you receive an error message read it and modify your code corresponding to it.

If you changed your mind and want to restore the handler to its original version click on the **Revert to original** button.

To test your code you have to restart Observer. This can be done by clicking on the **Restart NETAVIS Observer** button.

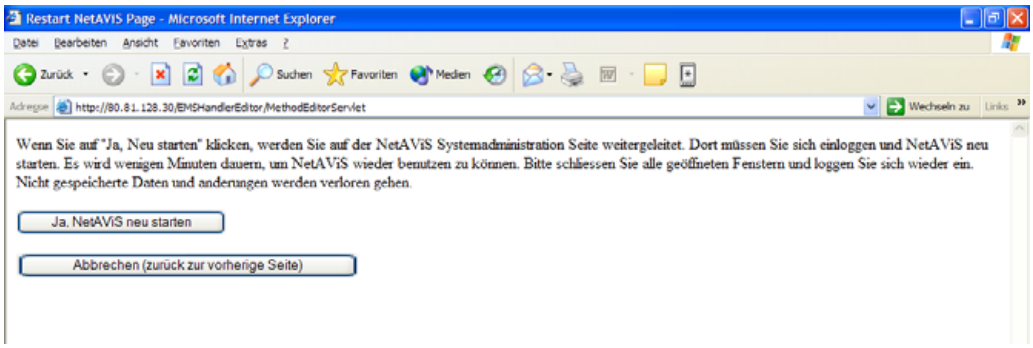


Abb. 2-3: Observer Handler Editor – Activating changes

If you answer with clicking on the **Yes, restart…** button then you will be redirected to the **webmin** interface where stopping/starting or Observer can be done. Clicking the **Cancel** button will navigate you back to the editor page.

2.3. Examples

Bringing you examples will help the most in understanding.

2.3.1 Handling of custom events

Apart from standard events, Observer provides a special event for third party event writers (e.g. working via the SNAP interface). Such a special event is called custom event which is similar to other events but its name and embedded instance variables are user definable. Such an event can have five String parameters. What is exactly stored in these parameters is up to the implementer. Of course, when numbers are stored as strings, the user has to convert them back to numbers before use.

Let' have an example now with custom events. Take that we have registered a custom event in the EMS as follows:

- name (type) of the event is “KapschSTSAlert”

- in the first parameter we will place the alert's type, such as: "service lane", "car crash"

We want to write a handler which will send out messages if the event was generated locally (by the server machine where your handler is running):

- sends an SMS to a given phone with the appropriate message
- pops up the online view of the camera on a Observer client terminal which is on a specific IP address.

We have to edit the method **handleEvent** of the class **CustomEventHandler** as follows:

```
    public void handleEvent(Event event) throws Exception {
        log("CustomEvent event handler starts");
        CustomEvent origEvent = (CustomEvent)event;
        if (origEvent.getCustomType().equals("KapschSTSAlert")) {
            if (event.isEventLocal())
                handleSTSEvent(origEvent);
            else
                log("Event is remote – skip handler body");
        }
        log("CustomEvent event handler ends");
    }

    private void handleSTSEvent(CustomEvent event) {
        ActionPropagator.sendSMS(event, "+36301234567", 0,
            event.getParam1(), null);
            ActionPropagator.clientControl(event,
                new TargetSelector("192.168.7.13", null,
                       TargetSelector.ONLINE_MONITOR, null),
                new TargetAction(event.getCameraID(), Actions.SHOW_LIVE_STREAM,
                       Modes.POP_IF_EXISTS, null));
}
```

2.3.2 Commend SoftVideo Interface

We have already covered some aspects of the Commend telephone centre in the plugin customization chapter (see page 9). With this example we will show you how this plugin (events generated by this plugin) can be used to start our SoftVideo application. Our components are as follows:

- The SoftVideo application, which is a Windows client listening on a given IP port and drawing images – received on the port - on the images client display.
- The Commend centre, to which one can connect calling devices attached to doors. Activating these devices will cause the centre to send trunk information about the calling device over its serial line.
- We have guards or secretaries sitting at monitors equipped with SoftVideo clients.

Our task is to bring life video streams from cameras attached to doors when the "doorbell" was activated.

What you have to start with is to check that the Commend plugin is activated and has the proper serial line setup (of course the centre has to be actively connected to the serial line and your server machine, too).

As we have already mentioned the Commend telephone centre can send control information (with a predefined protocol) which can tailor the behaviour of Observer. In out example we will define only two commands:

- Switch image stream of camera xx to monitor yyy, and

- Switch image stream of camera xx from monitor yyy.

The above mentioned monitor is a PC which can be uniquely identified by its IP address and port. To solve our problem we have to know that plugins generating so called custom events, which are handled in the class **CustomEventHandler**. During our work we will edit only this file (either directly with **vi** or via the web interface).

We will create two mappings, one for mapping a Commend camera to a Observer camera, and a second for mapping a monitor to an IP address. The maps will be stored in **Hashtable**s. Let's start writing the code:

```
public void handleEvent(Event event) throws Exception {
    log("CustomEvent event handler starts");
    CustomEvent origEvent = (CustomEvent)event;
    if (origEvent.getCustomType().equals("CommendGeneric")) {
        if (event.isEventLocal())
            handleCommendEvent(origEvent);
        else
            log("Event is remote - skip handler body");
    }
    else if (…)
    …
    log("CustomEvent event handler ends");
}

private void handleCommendEvent(CustomEvent event) {
    CommendMessage cmsg = new CommendMessage(event.getParam1());
    if(cmsg.getFieldPP().equals("5E")) {
        // Start/stop stream to external video viewer application
        process_5E_CommendMessage(cmsg, event);
    }
    else {
        log("WARNING: unhandled Commend message received: " + event.getParam1());
    }
}

private void process_5E_CommendMessage(CommendMessage cmsg, CustomEvent event) {
    log("process_5E_CommendMessage started");
    //-- MODIFY THE NEXT SECTION TO SUTE YOUR CONFIGURATION
    // - the Commend camera ID to NETAVIS camera ID and
    // - the stream target's IP address and port to which the stream
    //   is routed.
    if (fSourceMap == null) {
        fSourceMap = new Hashtable();
        fDestinationMap = new Hashtable();
        // This is an example line for setting up a mapping between
        // Commend camera IDs and NETAVIS camera IDs.
        // This first parameter is the Commend camera ID while the
        // second one is the NETAVIS camera ID.
        fSourceMap.put("0001", new NetavisCamera(3));

        // This is an example line for setting up a mapping between
        // stream target ID and its IP address and port to which
        // the image will be routed.
        // The first parameter is the ID of the monitor (or any other
        // output device) while the second parapeter defines the
        // the IP address and port of the destination.
        fDestinationMap.put("0003", new StreamTarget("192.168.7.30", 11112));
    }
    //-----------------------------------------------------------------
    String cmd;
    int cameraID = 0;
    if (cmsg.getFieldType().equals("01"))
        cmd = "START";
    else
        cmd = "STOP";

    String sourceCamera = cmsg.getFieldParam1();
    NetavisCamera nc = (NetavisCamera)fSourceMap.get(sourceCamera);
    if (nc != null)
        cameraID = nc.CameraID;
    String monitor = cmsg.getFieldParam2();
```

```
    StreamTarget st = (StreamTarget)fDestinationMap.get(monitor);
    if (st == null) {
       log("\tprocess_5E_CommendMessage: ERROR: No monitor was found for: " +
          monitor);
    }
    else {
       ActionPropagator.imageStreamControl(event, cameraID,
          st.IPAddress, st.Port, "JPEG", 5.0F, "MEDIUM", "MEDIUM", cmd);
       log("process_5E_CommendMessage finished OK");
    }
}

private class NetavisCamera {

    public int                CameraID;

    public NetavisCamera(int cid) {
       CameraID = cid;
    }
}

private class StreamTarget {

    public String       IPAddress;
    public int          Port;

    public StreamTarget(String ip, int port) {
       IPAddress = ip;
       Port = port;
    }
}

private class CommendMessage {
    private      String fFieldRR;
    private      String fFieldPP;
    private      String fFieldTT;
    private      String fFieldParam1;
    private      String fFieldParam2;
    private      String fFieldType;

    public CommendMessage(String fields) {
       StringTokenizer st = new StringTokenizer(fields,":");
       fFieldRR = st.nextToken().toUpperCase();
       fFieldPP = st.nextToken().toUpperCase();
       fFieldTT = st.nextToken().toUpperCase();
       fFieldParam1 = st.nextToken().toUpperCase();
       fFieldParam2 = st.nextToken().toUpperCase();
       fFieldType = st.nextToken().toUpperCase();
    }
    public String getFieldRR() { return fFieldRR; }
    public String getFieldPP() { return fFieldPP; }
    public String getFieldTT() { return fFieldTT; }
    public String getFieldParam1() { return fFieldParam1; }
    public String getFieldParam2() { return fFieldParam2; }
    public String getFieldType() { return fFieldRR; }
}
```

To activate and test the code please follow the instructions described in the section *Editing event handlers* on page 7.

2.3.3 Digital camera output and GUI control

In this example we will generate:

- two 1 second long square waves on the output pin of camera having 6 as ID
- popup the online view of camera having 7 as ID in a large view on a client terminal where a user named “admin” is logged in

when a motion did happen on camera having ID 5. To make the modifications we have to edit the method **handleEvent** in the class **MotionDetectionHandler** as follows:

```
public void handleEvent(Event event) throws Exception {
    log(“MotionDetection event handler starts”);
```

```
    MotionDetection origEvent = (MotionDetection)event;
    if (origEvent.getCameraID() == 5) {
        ActionPropagator.setCameraOutput(origEvent, 6,
            0, "\\1000/1000\\1000/1000\\");

        ActionPropagator. clientControl (origEvent,
            new TargetSelector(null, "admin", TargetSelector.ONLINE_MONITOR,null),
            new TargetAction(7, Actions.SHOW_LIVE_STREAM,
                  Modes.SHOW_AS_LARGE, null));
    }
    log("MotionDetection event handler ends");
}
```

The „action“ string in the method **setCameraOutput()**is a special notation. It tells the camera to which state (low, high) it should set and for how long. The “/” character pulls the output to high while the “\\” pulls the output to low (please note that the backslash character is an escape character; therefore if one wants to represent the backslash itself, the character must be doubled). The numbers in between represent delays in milliseconds. The above used notation can be read as: pull pin down, stay for 1 sec, pull to high, stay 1 sec, do the same again and leave the line in down position. In other words create two one second long square waves.

2.3.4 Digital camera input handling and archiving control

This example shows you how can you:

- handle camera input pin changes and
- start archive recording

The event generated by the plugin is called **CameraIOStateChange** therefore its handler is the **CameraIOStateChangeHandler**.

```
public void handleEvent(Event event) throws Exception {
    log("CameraIOStateChangeHandler event handler starts");
    int cameraID = origEvent.getCameraID();
    // Check the cameraID which will trigger saving
    if (cameraID == 6) {
        int duration = -1;
        if (origEvent.getCurrentState() == 1)
            duration = 0;
        ActionPropagator.setArchivingState(origEvent, 6, "",
                  "JPEG", 10.0F, "MEDIUM", "MEDIUM", duration);
    }
    log("CameraIOStateChangeHandler event handler ends");
}
```

As you can see, we are starting the recording only when the pin on camera 6 goes to high and stop when it is again in low state. With such a code you can do archiving only for a well-defined period of time (e.g. when a door is open).

2.3.5 PTZ positioning of a camera upon motion

As a final example we take the following situation. We defined motion detection on camera 5 and we also have a PTZ capable camera in our system (with ID 6). The first camera (with ID 5) has a large field of view and when it detects a motion in (for this example’s sake) a sector called “upper left” we want that the PTZ camera turn and room onto this detail. Therefore we pre-programmed the PTZ camera and set a position for the same portion as seen from camera 5 as “upper left”. Let’s say that this is position ID 1. At the same time when the positioning starts we want to archive the images of this PTZ camera with medium size, medium quality and 10 frames per second for ten seconds. Let’s go now into the class **MotionDetectionHandler** and edit the method **handleEvent** as follows:

```
public void handleEvent(Event event) throws Exception {
    log("MotionDetection event handler starts");
    MotionDetection origEvent = (MotionDetection)event;
    if (origEvent.getCameraID() == 5 &&
        origEvent.getEventName().equals("upper left")) {
```

```
        ActionPropagator.setPTZPosition(origEvent, 6, 1, null);

        ActionPropagator.setArchivingState(origEvent, 6, "", "JPEG",
           10.0F, "MEDIUM", "MEDIUM", 10);
    }
    log("MotionDetection event handler ends");
}
```

3. Customizing external I/O plugins

Connections to special third party software (e.g. facility management subsystems), handling of external devices (e.g. SMS terminals, serial devices) are realized in Observer by a separate service (External IO service). Software components implementing the connections to these software or hardware vehicles within the External IO service are called **plugins**. In the current version of Observer plugins can not be written by end-users. Users can benefit only from those which are delivered as bundle in the product. This chapter describes how parameters to these plugins can be customized to fit customer needs.

Corresponding to the architecture described earlier this service will produce events for the EMS service (e.g. data read via the serial line) and also execute actions (e.g. send an SMS messages). To simplify the work of a plugin one could say: a plugins transforms external information to Observer events and internal actions to external commands.

All parameters related to plugins are stored in XML files on the server. For customization you have to download these files from the server, make appropriate modifications and upload them again.

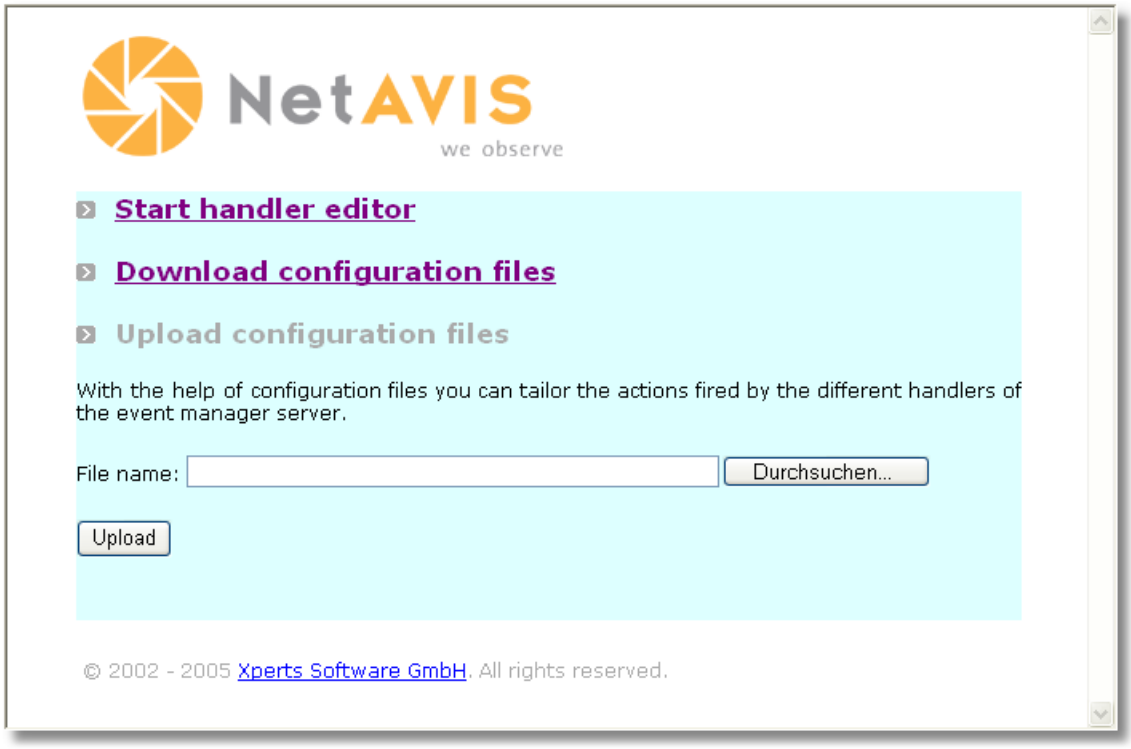
3.1. Download and upload of plugin configuration files

For down and uploading of XML data the **Customizer** web interface is used. This page is selectable via the standard web interface of a Observer server:

1. With your web browser connect to the Observer server.

2. Select your preferred language and click on **Start Customizer**.

3. Login with name: **admin** (default password: **admin** or type in your own if you changed it).
Now you see the **Customizer** page:



4. On the **Customizer** page click on **Download configuration files**.

5. Download the file named **common.utils.ExternalIOPluginMappings.default.xml** to your client.

6. Edit the file so that it contains the changes you wanted to make.

7. Upload the edited XML file back to the server (using the **Upload configuration files** part of the page).

8. Restart Observer (not the whole server) with the **[3] Stop NETAVIS** and **[2] Start NETAVIS** commands of the Admin interface (see the *NETAVIS Observer Server Installation and Administration* manual).

3.2. Customizing the plugin configuration XML file

3.2.1 Syntax of the file

If you are familiar with XML DTDs (Document Type Definition) the following description will give you the format and embedding of the plugin XML file. Fortunately, the elements and their attributes are self-explanatory.

```
<!DOCTYPE plugins [
    <!ELEMENT plugins ( plugin )* >
    <!ELEMENT plugin ( pluginClass , deviceHandlerClass? , deviceName? ,
deviceParameters? , flags? ) >
    <!ATTLIST plugin
        id CDATA #REQUIRED
        valid CDATA #REQUIRED
    >
    <!ELEMENT pluginClass ( #PCDATA ) >
    <!ELEMENT deviceHandlerClass ( #PCDATA ) >
    <!ELEMENT deviceName ( #PCDATA ) >
    <!ELEMENT deviceParameters ( #PCDATA ) >
    <!ELEMENT flags ( #PCDATA ) >
]>
```

After downloading the plugin XML file and having a look at it you will see that for each plugin a separate <plugin> element exists, defining different runtime parameters. After the <plugin> element two attributes are given. The “id=” attribute is used as an index and must be unique for each defined plugin in the XML file. The second is the “valid=” attribute which tells whether the given plugin should be activated or not at server start-up. If you set its value to “1” then the plugin is started, otherwise not.

The <pluginClass> element defines the name of the handler class. Please do not change it! The <deviceHandlerClass>, <deviceName> and <deviceParameters> elements are plugin-specific and their possible values depend on the purpose of the plugin.

3.2.2 Commend plugin

Commend is a telephone centre which can be controlled via a serial interface. Also, the centre can emit information via the serial line about its status changes or other events. When this plugin is enabled then incoming Commend commands are received and transformed into Observer events (about the CustomEvent which is generated by this plugin you can read later in this documents). The XML definition for the plugin looks as follows:

```
<plugin id="1" valid="0">
   <pluginClass>server.externalIO.custom_plugins.Commend</pluginClass>
   <deviceHandlerClass>server.externalIO.SerialIO</deviceHandlerClass>
   <deviceName>/dev/ttyS1</deviceName>
   <deviceParameters>9600;E;7;1;N</deviceParameters>
</plugin>
```

The <deviceName> element defines the serial port where the Commend centre is connected to. This is the line which is then read and written by the plugin. The <deviceParameters> element defines the serial line’s connection parameters, separated by semi-colons (“;”). The positions of these parameters are fixed, their possible values are as follows:

- Baud rate: 300, 600, 1200, 2400, 9600, 19200, 38400, 57600
- Parity: N = no, O = odd, E = even, M = mark, S = space
- Data bits: 5, 6, 7 or 8
- Stop bits: 1, 1.5 or 2
- Flow control: N = none, H = hardware (RTS/CTS), S = software (Xon/Xoff).

For a special use of the Commend centre Observer offers a so called SoftVideo solution. Please read about this feature in the **EMS Customization** chapter.

3.2.3 Camera PTZ (Pan/Tilt/Zoom) plugin

Positioning of PTZ camera heads are implemented in Observer as a plugin. Note: If you disable it by setting the valid=”0”you will not be able to use the PTZ feature of Observer! There are no tuneable parameters for this plugin because Observer handles all PTZ parameters internally. We mention it only for reference.

```
<plugin id="2" valid="1">
   <pluginClass>server.externalIO.PTZPlugin</pluginClass>
   <deviceHandlerClass/>
   <deviceName/>
   <deviceParameters/>
</plugin>
```

3.2.4 Camera I/O plugin

Some IP cameras have I/O ports which can be used for different purposes. An input connector e.g. can be used for connecting PIRs or other motion sensors to it, while output pins can be connected to electronic door openers. This plugin, when enabled will poll for input pin changes or set the output pin

to high or low values on user request. Note: If you disable it (valid=”0”) you will not be able to use this feature of Observer!

```
<plugin id="3" valid="1">
   <pluginClass>server.externalIO.CameraIOPlugin</pluginClass>
   <deviceHandlerClass/>
   <deviceName/>
   <deviceParameters/>
</plugin>
```

There is only one tuneable parameter for this plugin. Observer offers a special feature what is called “life signal”. If you connect a LED to one of your camera’s output pin, Observer will blink it until the services are running. You define the identifier of the camera as follows:

```
<deviceParameters>LifeSignal=camera_id</deviceParameters>
```

3.2.5 SMS plugin

Observer is able to send SMS messages via the Siemens M35 terminal. The device should be connected to the Observer server machine via the serial line.

```
<plugin id="4" valid="1">
   <pluginClass>server.externalIO.M35Sms</pluginClass>
   <deviceHandlerClass>server.externalIO.SerialIO</deviceHandlerClass>
   <deviceName>/dev/ttyS0</deviceName>
   <deviceParameters>9600;N;8;1;N</deviceParameters>
</plugin>
```

The <deviceName> element defines the serial port where the M35 terminal is connected to. This is the line which is then read and written by the plugin. The next element (<deviceParameters>) defines the serial line’s connection parameters, separated by semi-colons (“;”). Definition of these parameters is identical with the parameters described for the Commend plugin.

3.2.6 Keba Pasador plugin

This plugin interfaces Observer with a card reader device used at in-door ATM terminals for opening the door. The device can be connected to Observer via the serial line. When a card is drawn through the device, it sends time and card number information to Observer, which in turn generates a CustomEvent. In the EMS Customizing chapter you will learn how this event can be handled and how actions upon receipt can be triggered (e.g. start recording when the door was opened with a card). The plugin is defined as follows:

```
<plugin id="5" valid="0">
   <pluginClass>server.externalIO.custom_plugins.KebaPasador</pluginClass>
   <deviceHandlerClass>server.externalIO.SerialIO</deviceHandlerClass>
   <deviceName>/dev/ttyS0</deviceName>
   <deviceParameters>9600;N;8;1;N</deviceParameters>
</plugin>
```

The <deviceName> element defines the serial port where the Pasador card reader is connected to. This is the line which is then read by the plugin. The next element (<deviceParameters>) defines the serial line’s connection parameters, separated by semi-colons (“;”). Definition of these parameters is identical with the parameters described for the Commend plugin.

3.2.7 GEMOS plugin

GEMOS is a facility management system which can be coupled to a Observer server via this plugin. The plugin enables Observer to exchange status information with GEMOS. The incoming events generate CustomEvents which can be used to tailor Observer component behaviours.

```
<plugins>
   <plugin id="6" valid="0">
          <pluginClass>server.externalIO.custom_plugins.GEMOS</pluginClass>
          <deviceName>192.168.7.14:12000</deviceName>
          <deviceParameters>netavis passwd</deviceParameters>
   </plugin>
</plugins>
```

The <pluginClass> element defines the name of the handler class. Please do not change it. The <deviceName> element defines the IP address and port of the GEMOS server. Please use a colon (:) for separating the IP address (or name) of the server from the port. The <deviceParameters> element defines the login parameters of Observer. It is a space separated list of two values: GEMOS user name and password.

Setting up the plugin is only one part of the whole customization process. The next step is to identify all cameras in the Observer system and their corresponding GEMOS camera names (see externalID below). Observer identifier of a camera (see internalID below) is shown behind the camera name in the Observer Camera Administration application. This Observer camera ID number has to be mapped to GEMOS camera identifier.

When all ID information is available please download the file 'common.utils.ExternalDeviceMapping.default.xml', rename it to **common.utils.ExternalDeviceMapping.GEMOS.xml** and edit it to reflect the mapping.

Please note that a **camera** element MUST exist for all cameras which are relevant for the GEMOS system. When the GEMOS event handler in Observer cannot map a GEMOS camera ID to a Observer ID it will generate an error event instead of a video stream redirect event.

```
<mapping>
   <camera externalID="K:1" internalID="8" />
   <camera externalID="K:2" internalID="19" />
   <camera externalID="K:3" internalID="21" />

   <monitor externalID="M:1" internalID="1" />
</mapping>
```

Each monitor which can be assigned in GEMOS MUST also be defined here. In Observer there are no predefined clients (monitors) therefore this mapping is virtual only. This mapping is necessary for the sake of GEMOS, if there are no monitors in the system, it can not produce redirect events! Even if you define only one monitor you have to do it!

Our next XML file is the **server.event_manager.ShowLiveStreamDispatcher.default.xml**. This file contains mapping information for redirecting a given camera's video stream to different monitors in the Observer system. After downloading this file you have to rename it to **server.event_manager.ShowLiveStreamDispatcher.GEMOS.xml** before starting to edit it. The content of this file is as follows:

```
<showlivestream>
   <camera-rule internalID="8">
      <target type="ip">
         <name>192.168.7.13</name>
         <mode type="showinview">
          <viewname>panel</viewname>
         </mode>
      </target>
   </camera-rule>
   <camera-rule internalID="15-19">
      <target type="user">
         <name>admin</name>
         <mode type="showaslarge"/>
      </target>
   </camera-rule>
   <camera-rule internalID="all">
      <target type="ip">
         <name>192.168.7.13</name>
         <mode type="popifexsists"/>
      </target>
   </camera-rule>

<!--
   <monitor-rule internalID="1">
      <target type="ip">
         <name>192.168.7.13</name>
         <mode type="showinview">
          <viewname>panel</viewname>
         </mode>
      </target>
   </monitor-rule>
-->
</showlivestream>
```

You can make mapping rules for cameras or for monitors. If you want to make a mapping for a camera, please use the Observer internal camera ID in the 'camera-rule' element's 'internalID' attribute. The 'name' element defines the IP address of the monitor where the stream of this camera will be redirected. The mode can also be tailored using the 'mode' and 'viewname' elements. In a 'camera-rule' you can have more 'target' elements defining more than one target. Please refer the DTD part of the XML files for valid attribute values.

As an altervative to camera-rule, you can use mapping between GEMOS monitor ID (see externalID attribute of the 'monitor' tag in 'ExternalDeviceMapping' XML file) and a Observer monitorID, which is defined by the attribute 'internalID' in the 'camera' tag 'ExternalDeviceMapping'.

The same internalID has to be used in the 'monitor-rule' in 'ShowLiveStreamDispatcher'. Via 'monitor-rule' the GEMOS system advises Observer where (see 'target type' attribute) and how (see 'mode' tag) to show live stream of the camera.

The mapping policy is as follows:

- When a rule exists for a camera it will be used
- when no rule exists for a camera and the GEMOS monitor ID can be mapped to a Observer virtual monitor ID and a rule exists for this ID it will be used

When you are finished with the editing upload the renamed files onto the server. These XML files will automatically be reloaded by the handler, so after modification there is no need for server restart. Any time you need a modification simply download the needed file, edit it then upload it to the server. Please make sure to use the correct names for the above mentioned XML files otherwise the servers and handler plugins will not find them.